Exploiting Metrics to Facilitate Grammar Transformation into LALR Format

James F. Power
National University of Ireland, Maynooth
Computer Science Dept
County Kildare, Ireland
James.Power@may.ie

Brian A. Malloy
Clemson University
Computer Science Dept
Clemson, SC USA
malloy@cs.clemson.edu

ABSTRACT

The parser-generator yacc is one of the oldest examples of a domain-specific language, providing significant enhancements over hand-coded parsers in the area of speed, efficiency and maintainability. Despite its widespread use, often in highly complex systems such as compilers or program analysis tools, there is relatively little written about the integration of parsing, and yacc-based parsers in particular, into the software engineering process.

We exploit software metrics as an aid toward estimating the complexity of preparing a grammar for the ISO C++ programming language for input to yacc. Our metrics provide a means of assessing the relative merits of the trade-off between preserving the grammar's structure and rearranging it to ease implementation of the resulting parser. We see this work as part of a larger process of designing well-engineered, re-usable and reliable program processors, which themselves will play an important role in the future design of code-based software-engineering tools.

1. INTRODUCTION

The history of parser construction stretches as far back as the establishment of software engineering as a discipline, if not further[8]. In spite of this history, there has been relatively little interaction between these two disciplines. Furthermore, the typical approach to parser development relies more on experience and ad hoc techniques than a formal, documented engineering approach. In spite of the existence of tools such as yacc[9], a domain-specific language for automatic generation of parsers, considerable effort is often required to convert the language specification to a suitable format for these tools.

Construction of a parser for a programming language, and in particular large-scale languages such as C++[6], push the problem into the domain of software engineering. Despite the widespread use of yacc, one of the oldest examples of a domain-specific language, there has been little written about the integration of parsing, and yacc-based parsers in particular, into the software engineering process.

In this paper, we describe an approach for integrating the task of parser construction into the software engineering process. We describe a prototype approach to developing an initial parser, that preserves the structure of the grammar, using backtracking parsers or parsers with extended look-ahead. We then describe the use of software metrics to facilitate the transition from the prototype to a working LALR parser[1]. The metrics can highlight difficult parts of the grammar that will require the most effort to transform into LALR form, and indicate parts of the program that will require more extensive testing to guarantee robustness of the final product. Finally, we describe three correctness preserving grammar transformations that not only facilitate the transition to a working LALR parser, but that also serve to document the correctness of the transformed grammar.

In the next section, we provide background about grammars and parsing and in Section 3 we describe our approach for integrating parser construction into the software engineering process. We describe the correctness-preserving transformations in Section 4, and develop metrics in Section 5. Finally, we draw conclusions in Section 6.

2. GRAMMARS AND PARSING

In this section we define some of the terminology associated with context-free grammars and LALR parsing. Context-free grammars, originally a theoretical formalism, are one of the earliest examples of domain-specific languages. Context-free grammars are used in the construction of editors and other tools, and are used to describe the syntax of programming and other formal languages. In the form of their implementations in parser generators such as yacc, context-free grammars are foundational in the engineering of compilers, program analysis tools and other program processors.

A general description of languages, context-free grammars and parsing can be found in reference [1].

Given a set of words, a *language* is defined as a set of valid sequences of these words. A *grammar* defines a language; any language can be defined by a number of different grammars. When describing formal languages such as programming languages, we typically use a grammar to describe the *syntax* of that language; other aspects, such as the semantics of

[1]The actual algorithm used by yacc is based on LALR(1) parsing; in this paper, for simplicity, we refer to *LALR(1) parsing* as *LALR parsing*

SAC 2001 Las Vegas, NV

Accepted for the 16th ACM Symposium on Applied Computing
Las Vegas, USA, March 11-14, 2001

the language typically cannot be described by context-free grammars.
A grammar is defined over a set of vocabulary symbols, partitioned into *terminal* and *non-terminal* symbols, with one distinguished non-terminal symbol called the *start* symbol. The grammar defines a language using a set of *production rules*, each of which consists of a non-terminal on the left, and a sequence of vocabulary symbols on the right hand side. These production rules, when read as left-to-right rewrite rules, define the language consisting of all those sequences of terminal symbols that can be generated from the start symbol.
The procedure of using a grammar to derive a sentence in its language, known as *parsing*, can proceed in a top-down or bottom-up manner. The typical bottom-up approach, known as a *shift-reduce parser* works as follows. Terminals from the sentences are shifted onto the parser's stack until a sequence matching the right hand side of a production rule is found. The production rule is then applied by replacing the sequence corresponding to its right hand side with the non-terminal on the left of the rule, a process known as reduction. This process continues until all symbols have been consumed, and the entire sequence has been reduced to the start symbol.
Automating the process of bottom-up parsing involves constructing an automaton which recognizes right hand sides and then applies the appropriate reductions. Since a grammar may be ambiguous, and, indeed, some languages cannot be represented by an unambiguous grammar, it is not always possible to determinize this process. This nondeterminism is reflected in a choice of actions being presented to the parser: a choice between shifting a symbol and reducing by a rule, known as a shift-reduce conflict, and a choice between reductions using two different rules, known as a reduce-reduce conflict.
The traditional approach to bottom up parsing uses one symbol of lookahead from the sentence being parsed to guide the choice of action. The most common bottom-up parsing algorithms using one symbol of lookahead are the SLR(1), LR(1) and LALR(1) algorithms, which are described in detail in most compiler texts, including reference [1].
The parser generator yacc[2] takes a context-free grammar and generates the C source code for an LALR parser. Central to this parser is a deterministic finite-state automaton that recognizes right hand sides. If the grammar is ambiguous, or not amenable to determinization using the LALR algorithm, this will be manifested in terms of shift-reduce and reduce-reduce conflicts in the states of this automaton.

3. SOFTWARE ENGINEERING APPROACH

The history of parser construction stretches as far back as the establishment of software engineering as a discipline, if not further. In spite of this history, there has been relatively little interaction between these two disciplines. Furthermore, the typical approach to parser development relies more on experience and ad hoc techniques than a formal, documented engineering approach. In spite of the existence of tools such as yacc, which automate the task of parser generation, considerable effort is often required to convert the language specification to a suitable format for these tools. Compiler texts, due to space constraints, typically present academic examples demonstrating particular transformations and techniques. However, construction of a parser for programming languages, and in particular large-scale languages such as C++, push the problem into the domain of software engineering. Furthermore, there are two properties of the parser development process that make it particularly amenable to software engineering methodology.

- First, the software engineer may not have to develop an initial specification based on informal requirements, but rather is provided with a full and concrete definition of the grammar, usually in the standards document. In some cases, for example C and Java, the grammar is already in a parsable form. However, in the case of ansi C++, the grammar as presented in Appendix A is not easily converted to LALR form: containing "constructs that are hard to fit into a yacc grammar", as stated in reference [16, Section 3.3.2].
- Second, the specification is presented as a formal grammar so that the syntax of the language is defined unambiguously. This level of formality is contrary to the usual experience of the software developer who must typically communicate with the client to develop the software specification towards this level of formality. Common sense would suggest that the parser developer should exploit this advantage.

From these two properties, it is possible to identify at least three main areas of application of software engineering techniques. These three areas involve the use of the grammar initially for developing a prototype of the parser, as a source of complexity metrics to guide development and as a basis for the application of a sequence of correctness preserving transformations towards an LALR parser.
Typically, a *prototype* will implement the software specification, not necessarily efficiently, in an effort to quickly provide an initial working model. Two of the main functions of a prototype are relevant for parser development. First, the prototype can act as a yardstick against which the correctness of the ultimate working version can be measured. Second, the prototype can support modular development by allowing construction of other related components to proceed, pending the prototype's replacement by the actual working program.
In contrast to the transformations required to construct an LALR parser, a prototype should involve minimal changes to the original grammar. The use of either backtracking parsers, such as byacc, or parsers with extended look-ahead capabilities, such as ANTLR or javacc, allow the user to sacrifice efficiency in the interests of preserving the original grammar structure[2; 11; 13]. Even though these tools can be used for the construction of efficient parsers, they do not *require* one symbol lookahead, as is the case with the yacc family of parsers.
The developer can derive confidence in the correctness of the prototype parser from its similarity with the original grammar, and eventually this prototype can be used as an oracle against which the correctness of the final LALR parser can be measured. Furthermore, semantic actions can be added to the prototype and, as the grammar is transformed

[2]In this paper, we use the term *yacc* to cover both the original yacc parser-generator, and more recent variants, such as bison and byacc, which are based on the same LALR parsing algorithm

during the development of the LALR parser, the resultant impact on the semantics can be taken into account.

For a large scale software development effort, **software metrics** can facilitate the transition from the prototype towards the working software. In particular, software metrics can serve as an indication of areas of difficulty that may slow the development process. Furthermore, metrics can highlight complex parts of the program that will require more extensive testing to guarantee robustness of the final product. In Section 5 we discuss metrics relevant in the development of an LALR parser.

Since both the original grammar and the final yacc source of the LALR parser constitute a formal specification of the language, it provides a framework for a series of **correctness preserving transformations** from the former to the latter. This parallels the use of formal methods in software engineering where one can use either program derivation in a step-by-step manner, or program verification on the final product. Correctness preserving grammatical transformations can support either derivation or verification and serve to reinforce confidence in the correctness of the final product, as well as providing a source of documentation on the development process. These points are developed further in Section 4.

4. CORRECTNESS-PRESERVING TRANSFORMATIONS

Using the C grammar from reference [7], directly as input to yacc, will produce a single conflict resulting from the well-documented "dangling else" ambiguity. By contrast, naively using the C++ grammar from reference [6], directly as input to yacc, can produce as many as 1360 errors[3]. While yacc will generate a parse regardless of the number of conflicts, it is desirable to eliminate conflicts in order to have confidence in the correct operation of the parser.

Thus, the development of LALR parsers is typically characterized by the process of conflict elimination. This process naturally directs the developer toward a strategy of transformational programming where the goal of each transformation is to eliminate one or more conflicts and, preferably, avoid introducing new ones. While this transformational approach parallels well-recognized software engineering strategies, it has the additional benefit of providing a framework for formal verification[4].

Good software engineering practice dictates that a process consisting of a series of transformations be guided by a change control strategy and accompanied by suitable documentation[15]. However, many completed LALR parsers are presented "as is" without any such documentation, and thus the series of transformations applied cannot readily be reverse engineered from the parser. A parser developer working from a software engineering perspective should be expected to document the complete path of transformations from the original grammar to the resulting parser.

One additional advantage of such documented transformation is that they occur between two formal entities. Since both the source and the product of each transformation is a grammar rule or a series of grammar rules, it should be possible to formally verify the correctness of the transformation, i.e., that both the source and the product accepts the same language. Indeed, if all transformations applied in this process are instances of known correct transformations, then the designer can have full confidence in the correctness of the result.

Despite the formal basis of these transformations, there is no general algorithm for transforming an arbitrary grammar into an equivalent LALR parsable form. In fact, the more general problem of proving that two context free grammars accept the same language is undecidable[5, Theorem 8.12]. This means that the parser developer must rely on experience and known heuristics to guide the transformations.

Conflicts arise when the parser has a choice of action. Conflict resolution strategies thus concentrate on removing this choice, typically by searching backward from the conflict and introducing determinism at an earlier stage in the parse. Central to any conflict resolution strategy will be a library of basic transformations. Examples of some such transformations are described in general terms in reference [1] and more specifically in reference [14].

To demonstrate these transformations we present two examples of grammar conflicts and examine some possible approaches to their resolution. These approaches are instances of conflict resolution patterns, and they are based on examples from reference [9, Chapter 8].

```
rule : command opt_expr '(' identifier_list ')' ;
opt_expr : '(' expr ')' — /* blank */ ;
```

Figure 1: *Shift-reduce conflict.* This grammar segment illustrates a shift-reduce conflict resulting from a choice between regarding an open parenthesis as either the start of an optional expression, represented by `opt_expr`, or the parenthesis preceding an identifier list.

4.1 Limited lookahead

Figure 1 is an instance of a shift-reduce conflict resulting from a choice between regarding an open parenthesis as either the start of an optional expression, represented by `opt_expr`, or the parenthesis preceding an identifier list.

A solution to this conflict here is to transform the rules to those shown in Figure 2. This transformation is an instance of *inlining*, where a non-terminal is replaced by some or all of its definition. This replacement is guaranteed to be correctness preserving since the language generated by the transformation is unchanged. We might document such a transformation as

$$inline(r_1, N_1, r_2)$$

where r_1 and r_2 are grammar rules and where the non-terminal N_1 occurs on the right hand side of r_1 and r_2 is a definition of N_1.

There may be further conflicts in the resulting rules. For example, if `expr` can derive an identifier then the parser will be unable to distinguish such a parenthesized expression from an identifier list containing a single identifier. One solution to this conflict is to defer this choice to the semantic processing stage, thus allowing the parser to accept a superset of the language. This is an instance of a common transformation pattern called *widening*. We might document such

[3]This figure represents our own experience and is subject to variance, depending on the version of yacc used and the strategy employed for representing optionality.

rule : command '(' expr ')' '(' identifier_list ')' — command '(' identifier_list ')' ;

Figure 2: *Elimination of shift-reduce conflict.* A grammar transformation pattern, *inlining*, is used to eliminate the shift-reduce conflict illustrated in the previous figure. The non-terminal, `opt_expr`, in the conflicting grammar is replaced by its definition.

```
1        person : girl
2               | boy
3               ;

4        girl   : ALICE
5               | BETTY
6               | CHRIS
7               ;

8        boy    : ALAN
9               | BRIAN
10              | CHRIS
11              ;
```

Figure 3: *Reduce-reduce conflict.* This figure illustrates a reduce-reduce conflict resulting from two possible derivations of the terminal `CHRIS` from the non-terminal `person`. This conflict pattern is referred to as *overlap of alternatives*, since the alternative `CHRIS` occurs in both `girl` and `boy`.

a transformation as

$$widen(r_1, N_1, N_2)$$

where r_1 is a grammar rule, and where the occurrence of non-terminal N_1 on the right hand side of r_1 is replaced by non-terminal N_2.

4.2 Overlap of alternatives

Figure 3 is an instance of a reduce-reduce conflict resulting from two possible derivations of the terminal `CHRIS` from the non-terminal `person`. The strategy for conflict resolution here is to eliminate this choice of derivation. We hoist `CHRIS` upwards through the grammar to the right hand side of the non-terminal `person`, and then amend the rules for `girl` and `boy` accordingly. The result of this transformation is illustrated in Figure 4.

This strategy for conflict resolution actually involves a series of transformations. First, the non-terminals `boy` and `girl` are inlined in the definition of `person`. Second, the common productions involving the terminal `CHRIS` are merged into a single production. This transformation is an instance of a pattern referred to as *factoring*. Third, extraneous productions involving the other alternatives for `girl` and `boy` are deleted.

For simple examples of factoring, we might document this transformation as

$$factor(r_1, r_2)$$

where r_1 and r_2 are rules for the same non-terminal with identical right hand sides. More specific forms of factoring, such as left factoring would require a more detailed notation. This deletion of extraneous productions, while correct in this example, requires some care in general. Indeed this transformation can only be performed if no other rule produces `CHRIS` in a derivation involving `girl` and `boy`. Clearly the difficulty of performing this transformation is related to the size and complexity of the grammar.

```
1        person : girl
2               | boy
3               | CHRIS
4               ;
5        girl   : ALICE
6               | BETTY
7               ;
8        boy    : ALAN
9               | BRIAN
10              ;
```

Figure 4: *Elimination of reduce-reduce conflict.* The strategy for eliminating the *overlap of alternatives* conflict is to hoist `CHRIS` upwards through the grammar to the right hand side of the non-terminal `person` and amend the rules for `girl` and `boy` accordingly.

5. THE ROLE OF METRICS

The previous section presented a structured approach to the development of an LALR parser, placing this process in a software engineering context. In this section, we investigate the role of software metrics in guiding the process of conflict resolution.

In general, metrics can be used in a number of ways to aid the software development process. At the highest level, coarse-grained metrics can be used by project planners to estimate the overall difficulty of the task at hand, and implement suitable planning procedures based on these metrics. As the project proceeds, project managers can use metrics to gauge the progress of the development in terms of the reduction in complexity as identified by those metrics. At a fine-grained level, metrics can help the software developer to identify "hot spots" of program complexity requiring particular attention.

Since the process of parser development centers on grammar transformations, appropriate metrics are those concerned with the complexity of these transformations. The previous section identified some patterns of grammar transformations; the procedures corresponding to these patterns must drive the measurement process. A common thread in each of the transformations is the notion of hoisting rules and non-terminals through the grammar toward the source of conflict. The size of the distance of the "hoist" together with the number of other non-terminals involved, must necessarily contribute to the complexity of the transformation process.

As with any programming project, size is an important indicator of potential complexity. Reference [3] describes a wide range of metrics that can be used to measure program size complexity. In terms of grammars, the most obvious coarse-grained size metric is the number of production rules in the grammar. In terms of LALR parser construction, the most widely used metric is the number of states in the LALR(1) finite state machine that drives the parsing process. Some of these coarse-grained metrics may be derived directly from the yacc output; examples for C, C++ and Java parsers are illustrated in table 1.

Coarse-grained Metric	C	C++	Java
Number of Non-Terminals	118	233	147
Number of Rules	404	880	502
Number of LALR states	692	1668	777

Table 1: *Coarse-grained metrics for some of the GNU parsers.* The results depicted in this table list some figures obtained from the output file generated by running GNU bison over three of the parsers from gcc version 2.95.2

Even between grammars of comparable size complexity there can still be measurable factors to distinguish the ease with which grammar transformations can be applied. As can be seen from the discussion of conflict resolution in the previous section, inter-dependencies between non-terminals can complicate the transformation process. One existing metric used to determine these inter-dependencies is the tree impurity metric which measures the degree to which a program call graph resembles a complete graph. The tree impurity metric and other metrics applying directly to grammars are presented in reference [12].
We have already seen how grammar transformations involve hoisting non-terminals through production rules. The complexity of this process is affected by two attributes of the production rules involved:

- **depth** – a measure of the height of the tree rooted at the source of the conflict, spanning the affected production rules and terminating with the hoisted non-terminal. Since each node in this tree will necessarily be affected by the hoisting process, a hoist over a greater depth is more likely to increase the complexity of the transformation. This view of the hoisting process parallels the concept of a *definition-use path* as used in program optimizations and software testing[1; 10]
- **breadth** – a measure of the number of non-terminals that will be affected by the hoisting process. This breadth metric corresponds to branches of the tree not directly involved in the hoist itself. A large number of such branches in the tree indicates an increase in the complexity of the transformation. This parallels the *fan-out* metric discussed in reference [3] which measures the calling dependencies for a particular node in the call graph.

Software metrics can be used not only as a guide to determine the complexity of the transformation process, but also to facilitate software testing. Parts of the grammar of high complexity or transformations involving a high degree of complexity bear an increased likelihood of generating faults. These parts of the grammar may require a more intense testing effort and thus, software metrics can play a role in the development of a suitable test suite.

6. CONCLUDING REMARKS

In this paper we have described an approach for integrating the task of parser construction into the software engineering process. In particular, we have described a role for prototyping, correctness-preserving transformations and software metrics in this process. We see this work as part of a larger process of designing well-engineered, reusable and reliable program processors. These parser-based program processors play an important role in the software engineering process, specifically in the construction of tools for software design, testing, maintenance and re-engineering.

7. REFERENCES

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] C. Dodd and V. Maslov. Btyacc – backtracking yacc version 3.0. Technical report, Siber Systems, `http://www.siber.com/btyacc/`, 2000.

[3] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. Thomson Computer Press, first edition, 1996.

[4] J. P. Gibson, T. F. Dowling, and B. A. Malloy. The application of correctness preserving transformations to software maintenance. *Proceedings of the International Conference on Software Maintenance*, page (to appear), October 11–14 2000.

[5] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[6] ISO/IEC. *International Standard: Programming Languages - C++*. Number 14882:1998(E) in ASC X3. American National Standards Institute, 1998.

[7] B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice-Hall, 1978.

[8] D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.

[9] J. R. Levine, T. Mason, and D. Brown. *lex & yacc*. O'Reilly & Associates Inc., second edition, 1992.

[10] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[11] T. Parr. Antlr version 2.7.0. Technical report, Magelang Institute, `http://www.ANTLR.org/`, 2000.

[12] J. F. Power and B. A. Malloy. Metric-based analysis of context-free grammars. *Proceedings of the International Workshop on Program Comprehension*, page (to appear), 2000.

[13] S. Sankar, S. Viswanadha, and R. Duncan. Java compiler compiler version 1.1. Technical report, Metamata Inc., `http://www.metamata.com/JavaCC/`, 1999.

[14] J. J. Sarbo. Grammar transformations for optimizing backtrack parsers. *Computer Languages*, 20(2):89–100, 1994.

[15] S. R. Schach. *Classical and Object-Oriented Software Engineering*. McGraw-Hill, fourth edition, 1998.

[16] B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, first edition, 1994.